A Data Model and Query Language for Distributed Service Discovery

Wolfgang Hoschek
CERN IT Division
European Organization for Nuclear Research
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

Abstract

In a large heterogeneous distributed system spanning administrative domains such as a DataGrid, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This paper develops a suitable database and query model as well as a generic and dynamic data model for such database systems. Unlike in the relational model the elements of a tuple in our data model can hold structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. An individual tuple element may, but need not, have a schema (XML Schema), in which case the content must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. The concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven. A query is formulated against a global database view and is insensitive to link topology and deployment model. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Example service discovery queries are given. Three query types are identified, namely simple, medium and complex. An appropriate query language (XQuery) is suggested. The suitability of the query language is demonstrated by formulating the example prose queries in the language. Detailed requirements for a query language supporting service and resource discovery are given. The capabilities of various query languages are compared.

1 Introduction

In a large heterogeneous distributed system spanning administrative domains such as a DataGrid [1, 2, 3, 4], it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. Other examples are a (worldwide) service discovery infrastructure for a multi-national organization, a Peer-to-Peer (P2P) file sharing system, the Domain Name System (DNS), the email infrastructure, a monitoring infrastructure or an instant messaging and news service. As in a data integration system [5, 6, 7], the goal is to exploit several independent information sources as if they were a single source. This enables information discovery and collective collaborative functionality that operates on the system as a whole, rather than on a given part of it. In such a database system, the set of information tuples in the universe is partitioned over one or more nodes from a wide range of distributed system topologies, for reasons including autonomy, scalability, availability, performance and security. A database model is required that clarifies the relationship of the entities in a distributed system.

The distribution and location of tuples should be transparent to a query. However, in practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world. Both requirements need to be addressed by an appropriate query model.

A data model remains to be specified. It should specify what kind of data a query takes as input and produces as output. Due to the heterogeneity of large distributed systems spanning many administrative domains, the data model should be flexible in representing many different kinds of information from diverse sources, including structured and semi-structured data. The key problem then is:

- *In a large heterogeneous distributed system spanning many administrative domains, what kind of database, query and data model as well as query language can support simple and complex dynamic*

information discovery with as few as possible architectural and design assumptions? How can one uniformly support queries in a wide range of distributed system topologies and deployment models, while at the same time accounting for their respective characteristics?

This paper answers the above question by developing a suitable database and query model as well as a generic and dynamic data model. This paper is organized as follows. Section 2 develops a database and query model. The concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven. Section 3 proposes a generic and dynamic data model. The dynamic data model addresses dynamic state maintenance problems. Here a tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary content and allows for refresh of that content at any time. Section 4 suggests XQuery [8] as an appropriate query language. Section 5 formulates in prose a representative set of example service discovery queries. Three query types are identified, namely *simple, medium* and *complex*. The suitability of the query language is demonstrated by formulating the example prose queries in the language. Section 6 compares our approach with related work. Detailed requirements for a query language supporting service and resource discovery are given. The capabilities of various query languages are compared. Finally, Section 7 summarizes and concludes this paper.

2 Database and Query Model

Database Model. A distributed database framework is used where there exist one or more nodes. Each node can operate autonomously. A node holds a set of tuples in its database. A given database belongs to a single node. For flexibility, the databases of nodes may be deployed in any arbitrary way (*deployment model*). For example, a number of nodes may reside on the same host. A node's database may be co-located with the node. However, the databases of all nodes may just as well be stored next to each other on a single central data server. The database tuples may be dynamically (re) computed on each query. A database may be anything that accepts queries from the query model and returns results according to the data model (see below).

The set of tuples in the universe is partitioned over the nodes, for reasons including autonomy, scalability, availability, performance and security. Nodes are interconnected with links in any arbitrary way. A link enables a node to query another node. A *link topology* describes the link structure among nodes. The centralized model has a single node only. For example, in a service discovery system, a link topology could tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain. Figure 1 depicts three example link topologies, namely ring, tree and graph (Peer-to-Peer). Depending on the application context, all topologies have their merits and drawbacks.

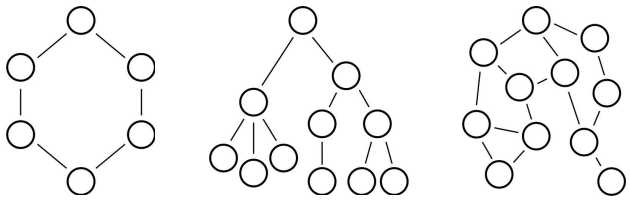

Figure 1: Ring, Tree and Graph Topology [9].

Query Model. Our query model is intended for read-only search. Insert, update and delete capabilities are not required and not addressed. We have defined these capabilities elsewhere [4, 10]. It is a general-purpose query model that operates on *tuples.* Discussion in this paper often uses examples where the term *tuple* is substituted by the more concrete term *service description.*

In practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world.

However, it is a strong user requirement that queries should be as insensitive as possible to any link topology and deployment model. In other words, a user should not need to reformulate a query when the node topology or deployment model changes, as is frequently the case in large distributed systems spanning many administrative domains such as P2P networks or cross-organizational Grids. A query model should not make any assumptions on the underlying database and query processing technology. P2P query engines, distributed database systems and centralized database systems should be able to answer the same queries. As in a data integration system, the goal is to exploit several independent information sources as if they were a single source. In support of these requirements, the concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven.

- **Query.** A query is formulated against a global

database view and is insensitive to link topology and deployment model. In other words, to a query the set of all tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples appears as a single, very large, table or XML document, respectively.

- **Query Scope.** The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Searching is primarily guided by the query. Scope hints are used only as necessary. A query is evaluated against a set of tuples. The set, in turn, is specified by the scope. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none.

A query scope is specified either *directly* or *indirectly*. For example, one can directly enumerate the tuples (service descriptions) to be considered. However, this is usually impractical. One can also indirectly define a query scope by specifying a set of nodes (or Internet domain names or table names), implying that the query should be evaluated against the union of all tuples contained in their respective databases. This corresponds to the concept of horizontally partitioned tables extensively used in large-scale relational database systems, in particular for distributed instances [11]. One can also indirectly specify the query scope by giving a time deadline, implying that as many tuples as possible should be considered, but only until the deadline has passed. Many more ways to specify a query scope can be envisioned. Both query and scope can prune the search space, but they do so in a very different manner.

3 Dynamic Data Model

Generic Data Model. In a large distributed system, a registry is populated from a large variety of heterogeneous remote data sources. The input and output of a query are instances of a generic data model, which in our case is XML based and models a document as a tree of nodes. XML [12] is used because one of its strengths is its flexibility in representing many different kinds of information from diverse sources, including structured and semi-structured data. XML is, above all else, a unifying integration technology.

The data model must be capable of modeling an XML document as well as a well-formed fragment of a document, a sequence of documents, or a sequence of document fragments. We note that there is no need to store tuples in XML; they just need to be presented this way, perhaps by middleware. For example, it is common to present data from relational databases, dynamic content generation systems and legacy command line tools as XML. A more sophisticated system can accept queries over an XML view and internally translate the query into SQL [13, 14, 15, 6].

The data model represents a set of tuples. A *tuple* has, unsurprisingly, zero or more XML attributes and zero or more XML elements.

In the relational model, a tuple has a number of column values. All tuples of all nodes are homogenous in the sense that their column values comply with a single strongly typed schema. In our model, this is not required. The elements (columns) of a tuple can hold structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. An individual tuple element may, but need not, have a schema (XML Schema [16]), in which case the content must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. An element (column) is typed (`type XML`), but obviously in a very loose manner. A tuple is a multi-purpose data container that may contain arbitrary content. Unlike in a RDBMS, a single (logical) tuple set contains *all* tuples. This implies that a query need not specify a "table" or "tuple set name" to indicate the type of tuples that should be considered. Rather, predicates within the regular query language are used to select the desired tuples from the single set. Arguably, it is more appropriate to adopt XML parlance and also use the term *element* instead of *tuple*. Nevertheless, continuing to use established terminology from the relational world seems to improve clarity more than it is misleading. Discussion in this paper often uses examples where the term *tuple* is substituted by the more concrete term *service description*.

The actual query is fed as input an XML representation that has the following form.

```
<tupleset>
   zero or more tuples go here
</tupleset>
```

The output of a predicate query (see below) is a subset of its input. The output of a constructive query (see below) is an arbitrary structure of the following form.

```
<tupleset>
   zero or more XML elements go here
</tupleset>
```

In any case, the query engine always encapsulates the query output with a `tupleset` root element. A user query need not generate this root element as it is implicitly added by the environment.

Dynamic Data Model (DDM). In a large distributed system spanning many administrative domains, a registry is populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. To address dynamic state maintenance problems, we propose the *Dynamic Data Model (DDM)*, which is an instantiation of the Generic Data Model. In DDM, a tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary *content* and allows for refresh of that content at any time, as depicted in Figure 2.

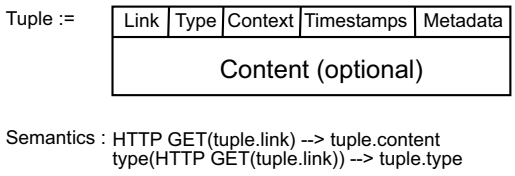

Figure 2: Tuple Link allows for Refresh of Tuple Content at any time.

Examples for content include a service description, file, picture, current network load, host information, stock quotes, etc. Content of a given *type* is maintained for a given *context* (purpose) and may optionally be associated with some arbitrary shaped *metadata*. A tuple and its content are valid for some time span only. At any time, the current (up-to-date) content can be retrieved from the authoritative content provider via a dynamic pointer called a *content link*. The pointer can be used if stale content is to be avoided. Detailed motivation and justification is given in our prior studies [10, 4].

The actual query is fed as input an XML representation that has the form depicted in Figure 3.

```
<tupleset TS4="100">
    <tuple link="http://sched001.cern.ch/getDescription"
           type="service" ctx="parent" TS1="10" TC="15"
           TS2="20" TS3="30">
       <content>
          <service>
            <interface type="http://cern.ch/Scheduler">
              <operation>
                <name>void submitJob(String job)</name>
                <allow> http://cms.cern.ch </allow>
                <bind:http verb="GET"
                 URL="http://sched.cern.ch/submitjob"/>
              </operation>
            </interface>
          </service>
       </content>
       <metadata>  <owner name="http://cms.cern.ch"/>
       </metadata>
    </tuple>

    <tuple link="http://repcat.cern.ch/pub/getDesc?id=4711"
           type="service" ctx="child" TS1="30" TC="0"
           TS2="40" TS3="50">
    </tuple>

    <tuple link="http://monitor.cern.ch/pub/getHostInfo"
           type="hosts" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <hosts>
             <host name="fred01.cern.ch" os="redhat 7.2"
                   arch="i386"  mem="512M"/>
             <host name="fred02.cern.ch" os="solaris 2.7"
                   arch="sparc" mem="8192M"/>
         </hosts>
       </content>
    </tuple>
</tupleset>
```

Figure 3: Dynamic Tuple Set.

4 XQuery Language

XQuery [8, 17, 18, 19] is the standard XML [12] query language developed under the auspices of the W3C. A number of excellent introductions and compact summaries of its features have already appeared [15, 8]. Therefore, here we only briefly mention some of the more interesting features.

XQuery is designed to be a small, easily implementable language in which queries are concise and easily understood. The language is derived from an XML query language called Quilt [20], which in turn borrowed features from several other languages. From XPath [21] and XQL [22] it took a path expression syntax suitable for hierarchical documents. From XML-QL [23] it took the notion of binding variables and then using the bound variables to create new structures. From SQL [24] it took the idea of a series of clauses based on keywords that provide a pattern for restructuring data (the SELECT-FROM-WHERE pattern in SQL). From OQL [25] it took the notion of a functional language composed of several different kinds of expressions that can be nested with full generality [8].

XQuery can dynamically integrate external data sources via the `document(URL)` function. The `document(URL)` function can be used to process the XML results of remote operations such as `getNetworkLoad`, invoked over HTTP. XQuery uses XPath [21] expressions for hierarchical navigation to parts of an XML document. The

`document("http://mysite.org/med.xml")/record` expression returns the ordered list of all `record` children of the root of the given document. `document("med.xml")//record` returns the list of `record` elements at any depth in the document. Path predicates, interspersed in path expressions, restrict the navigation; for example, `//entry[date="1/9/90"]` returns all the `entry` elements having at least one date child, whose string values is `"1/9/90"` [15].

XQuery provides the usual set of first-order operators (arithmetic, logical and set-oriented); the comma is a list concatenation operator. For example, `(//entry, //name)` returns the concatenation of the list of all entries, followed by all names, in document order. Second order operators in XQuery are the logical quantifiers `ANY`, `ALL`, and `SORT`. For example, `document("med.xml")//entry SORTY BY date DESC` will return all entry elements, the most recent first.

FLWR expressions (pronounced *"flower"*) consist of three parts: a `FOR-LET` clause that makes variables iterate over the result of an expression or binds variables to arbitrary expressions, a `WHERE` clause that allows specifying restrictions on the variables, and a `RETURN` clause that can construct new XML elements as output of the query. For example, the following query retrieves all the medical records of people with health problems that have been related to pollution within the last ten years [15]:

```
FOR $r in document("med.xml")//record,
    $e in $r/entry
WHERE $e/date > "1/1/90" and
    contains($e/diagnosis, "pollution")
RETURN <pollutionIncident>
          $r/@ssNo, $e/diagnosis
       </pollutionIncident>
```

5 Query Examples and Types

The suitability of the query language for service and resource discovery is now demonstrated by formulating example prose queries in the XQuery language. One can distinguish three types of queries: *simple, medium* and *complex*. The latter are more powerful than the former. Nevertheless, we will see that even a simple query is a powerful tool.

Simple Query. Simple queries are most often used for discovery. A simple query finds all tuples (services) matching a given predicate or pattern. The query visits each tuple (service description) in a set individually, and generates a result set by applying a function to each tuple. The function usually consists of a predicate and/or a transformation. Individual answers are added to a (initially empty) result set. An empty answer leaves the result set unchanged. A simple query has the following form:

```
R = {}
for each tuple in input
  R = R UNION { function(tuple) }
endfor
return R
```

Example simple queries are:

- *(QS1) Find all (available) services.*

```
RETURN /tupleset/tuple[@type="service"]
```

- *(QS2) Find all services that implement a replica catalog service interface that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN).*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE
  SOME $op IN $s/interface[@type = $repcat]/operation
  SATISFIES ($op/name="XML getPFNs(String LFN)" AND
             $op/bindhttp/@verb="GET" AND
             contains($op/allow, "http://cms.cern.ch"))
RETURN $tuple
```

- *(QS4) Find all local services (all service interfaces of any given service must reside on the same host).*

```
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE count(distinct(hostname($s/interface/operation/
            bindhttp/@URL))) <= 1
RETURN $tuple
```

- *(QS5) Find all services and return their service links (instead of descriptions).*

```
FOR $tuple in $doc/tupleset/tuple[@type="service"]
RETURN <tuple> {$tuple/attribute::*} </tuple>
```

- *(QS6) Find all CMS replica catalogs and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE
  SOME $op IN $s/interface[@type = $repcat]/operation
  SATISFIES ($op/name="XML getPFNs(String LFN)" AND
    $op/bindhttp/@verb ="GET" AND
    contains($op/allow, "http://cms.cern.ch"))
RETURN
```

```
FOR $pfn IN invoke($s, $repcat,
    "XML getPFNs(String LFN)",
    "http://myhost.cern.ch/myFile")
    /tupleset/PFN
WHERE starts-with($pfn, "ftp://")
RETURN $pfn
```

Note that all but the last two queries return service descriptions, whereas the latter return additional or entirely different information (service links or physical file names). We term the former queries *predicate (or filter) queries*. The structure of the result set is predetermined in the sense that query output must be a subset of query input. We term the latter queries *constructive queries*, because they construct answers of arbitrary structure and content. Predicate queries are a subset of constructive queries. A constructive query function that always returns `"Hello World"` or an empty string is legal, but not very useful.

Further, note that the last query involve multiple independent data sources and matches on dynamically delivered content (via remote invocation of operations `getPFNs`), rather than on values being part of service descriptions. We call these queries *dynamic queries*, as opposed to *static queries*. To support dynamic queries, a query language must provide means to dynamically retrieve and interpret information from diverse remote or local sources.

Dynamic queries can sometimes be reformulated as static queries. For example, the LFN/PFN database information of query *QS6* could be made available as part of the tuple set. In practice, this is typically infeasible for reasons including database size, consistency, information hiding, security and performance. Publishing highly volatile attributes such as CPU load as part of tuples leads to stale data problems. Clearly dynamic invocation is a more appropriate vehicle to deliver CPU load. Alternatively, custom push protocols can be used, for example as defined in the Grid Monitoring Architecture [26].

Medium query. A medium query computes an answer over a set of tuples (service descriptions) as a whole. For example, it can compute aggregates like number of tuples, maximum, etc. Example medium queries are:

- *(QM1) Find the CMS storage service with the largest network bandwidth to my host "dummy.cern.ch" (assuming there exists a service estimating bandwidth from A to B).*

```
LET $source := "dummy.cern.ch"
LET $storage := "http://cern.ch/storage-1.0"
LET $sorted := /tupleset/tuple[@type="service" AND
         content/service/@owner="cms.org" AND
         content/service/interface/@type=$storage]
         SORTBY (bandwidth($source, host(./@link)))
RETURN $sorted[last()]

DEFINE FUNCTION bandwidth($src, $dst) {
   document("http://net.cern.ch/estimate?src=",
            $src,"&dst=",$dst)
}
```

- *(QM2) Return the number of replica catalog services.*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
RETURN count(/tupleset/content/service
             [interface/@type=$repcat])
```

- *(QM3) Find the two CMS execution services with minimum and maximum CPU load and return their service description and load.*

```
LET $exec := "http://cern.ch/executor-1.0"
LET $tuples := /tupleset/tuple[@type="service" AND
              content/service/@owner="cms.org" AND
           content/service/interface/@type=$exec]]
LET $sorted := FOR $tuple IN $tuples RETURN
  <item>
    {$tuple}
    <load>
      {invoke($tuple/content/service, $exec,
               "String cpuLoad()", "")}
    </load>
  </item> SORTBY (load)
RETURN
  <min> $sorted[1] </min>
  <max> $sorted[last()] </max>
```

The query is applied to the set as a whole. For example, *QM3* is interesting in that it involves crossing tuple boundaries, which simple hierarchical query languages typically do not support. Like a simple query, a medium query can be static or dynamic. It can be a predicate query or a constructive query.

Complex query. Complex queries are most often used for advanced discovery or brokering. Like a medium query, a complex query computes an answer over a set of tuples (service descriptions) as a whole. However, it has powerful capabilities to combine data from multiple sources. For example, it supports all database join flavors. Like any other query, a complex query can be static or dynamic. It can be a predicate query or a constructive query. Example complex queries are:

- *(QC1) Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

```
LET $exeType := "http://cern.ch/executor-1.0"
LET $stoType := "http://cern.ch/storage-1.0"
```

```
FOR $exec IN /tupleset/tuple[content/service/
                  interface/@type=$exeType],
    $stor IN /tupleset/tuple[content/service/
                  interface/@type=$stoType
                  AND domainName(@link) =
                  domainName($exec/@link)]
RETURN
  <pair>
    {$exec}
    {$storage}
  </pair>
```

- *(QC2) Find all hosts that run more than one replica catalog with CMS as owner. (Want to check for anomalies).*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
LET $hosts := /tupleset/tuple/hostname(@link)
    [content/service[interface/@type = $repcat AND
     content/service/@owner = "cms.org"]]
FOR host IN $hosts
RETURN <host> {$host} </host>
WHERE count(hosts[./ = $host]) > 1
```

6 Related Work

Let us assess the suitability of the query capabilities of various query languages in the context of service and resource discovery.

LDAP. The Lightweight Directory Access Protocol (LDAP) [27] inherits its query and data model from X.500 [28]. The data model is not dynamic and it is not XML based. No example service discovery query except *QS1* and *QS5* can be expressed with the LDAP query language. This would also be the case if LDAP were defined on an XML data model. An example system using this query and data model is the Metacomputing Directory Service (MDS) [29, 30].

The data model is based on the *entry*, which contains data about some object (e.g. a person). An entry is composed of attributes, which have a type and one or more values. The attribute type determines what kinds of values are legal. An entry has a mandatory attribute, which is a hierarchical identifier termed *distinguished name* (*DN*). An example DN is `cn=Barbara Jensen, o=University of Michigan, c=US`. Because of its hierarchical nature, a DN can be seen as organizing a set of entries into a tree structure, the *Directory Information Tree* (DIT). Usually the tree is organized according to political, geographical, or organizational boundaries. For comparison, an HTTP URL with the usual attribute-value pairs can be considered equivalent to a DN. An XML tuple with a content link corresponds to an LDAP entry. A set of such tuples corresponds to a set of LDAP entries. Both sets can be interpreted as a tree. Operations are provided to query, add, modify, and delete entries from the tree.

The LDAP query language has the following capabilities. A query returns a set of matching entries. A query can specify a base DN, scope, filter, timeout, maximum result set size and the names of attributes to return for each matching entry. The base DN decides the position in the name space tree at which the search should be started. An empty string implies starting at the root of the tree. The scope flag indicates which entries should be considered: just the base DN entry, all immediate descendent entries of the DN, or all entries at or below the DN. The filter is applied to each entry selected by the scope. A filter is an expression that logically compares (`=, <=, >=`) the string value of an attribute (`email`) with a string constant, optionally with a substring match joker (`picture*.jpg`) and approximate string equality test ($\sim$). Filters can be combined with Boolean `AND, OR` and `NOT` operators. For example, the query `o=anl.gov,c=US??persons?(&(cn=Mark*)(sn=G*))` returns every person entry whose name starts with Mark and whose surname starts with "G".

Clearly the expressive power of this language is insufficient for service and resource discovery use cases and most other non-trivial questions.

SQL. The relational data model is well suited for static centralized systems. However, it is unsuitable for a large distributed system spanning many administrative domains, populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. The relational data model does not allow for semi-structured data. All tuples must be homogenously structured in the sense that their column values comply with a strongly typed schema. Tuples with different schema belong to different tables. Hence, a query cannot operate on a single (logical) set containing *all* tuples. A query must have out-of-band knowledge of the relevant table names and schemas, which themselves may not be heterogeneous but must be static, globally standardized and synchronized. This seriously limits the applicability of the relational model in the context of autonomy, decentralization, unreliability and frequent change.

SQL [24] is a rich and expressive general-purpose language defined over the relational data model. In addition to the above limitations, SQL lacks hierarchical navigation as a key feature and other capabilities such as dynamic data integration, expression nesting with full generality as well as regular expression matching. As a result, some example queries cannot be expressed (e.g. *QS6, QM1, QM3*) and most can only be

expressed with extremely complex queries over a large number of auxiliary tables [15, 4]. The same holds for inserts and updates.

The relational data model and SQL are, for example, used in the Relational Grid Monitoring Architecture (RGMA) system [31] and the Unified Relational GIS Project [32].

Other. None of the example discovery queries can be satisfied with a lookup by key (e.g. globally unique name). This is the type of query assumed in most P2P systems such as DNS [33], Gnutella [34], Freenet [35], Tapestry [36], Chord [37] and Globe [38], leading to highly specialized *content-addressable* networks centered around the theme of distributed hash table lookup. Note further that almost no queries are exact match queries (i.e. given a flat set of attribute values find all tuples that carry exactly the same attribute values), assumed in systems such as SDS [39] and Jini [40]. They are also not fuzzy keyword searches, as used in web search engines. Next, queries do not specify that at most one result should be returned.

The limited expressiveness of the above mentioned query languages allows for easy implementation and some straightforward optimizations, but it also dramatically limits their applicability and ability to cope with changing requirements, leading to a flurry of very similar but not identical special-purpose systems, each supporting yet another narrow custom query type. These systems may well serve a special-purpose important for a given niche, but are unsuitable for supporting service discovery, let alone ubiquitous service and resource discovery for a wide range of applications and user communities.

The greater the number and heterogeneity of content and applications, the more important expressive general-purpose query capabilities become. Clearly realistic ubiquitous service and resource discovery *stands and falls* with the ability to express queries in a rich general-purpose query language. More precisely, a query language suitable for service and resource discovery should meet the requirements stated in Figure 4 (in decreasing order of significance). As can be seen from the table, LDAP, SQL and XPath do not meet a number of essential requirements, whereas the XQuery language meets all requirements and desiderata posed.

7 Conclusions

This paper develops a database and query model as well as a generic and dynamic data model that address the problems of large heterogeneous distributed system spanning many administrative domains. Unlike in the relational model the elements of a tuple in our data model can hold structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. An individual tuple element may, but need not, have a schema (XML Schema), in which case the content must be valid according to the schema. The elements of all tuples may, but need not, share a common schema. The concepts of (logical) query and (physical) query scope are cleanly separated rather than interwoven. A query is formulated against a global database view and is insensitive to link topology and deployment model. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model.

Example service discovery queries are given. Three query types are identified, namely *simple, medium* and *complex*. An appropriate query language (XQuery) is suggested. The suitability of the query language is demonstrated by formulating the example prose queries in the language. Detailed requirements for a query language supporting service and resource discovery are given. The capabilities of various query languages are compared.

References

[1] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. Journal of Supercomputer Applications*, 15(3), 2001.

[2] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[3] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.

[4] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery.* PhD Thesis, Technical University of Vienna (submitted), 2002.

[5] J.D. Ullman. Information integration using logical views. In *Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[6] Daniela Florescu, Ioana Manolescu, Donald Kossmann, and Florian Xhumari. Agora: Living with XML and Relational. In *Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, February 2000.

[7] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

Capability	XQuery	XPath	SQL	LDAP
Simple, medium and complex queries over a set of tuples	yes	no	yes	no
Query over structured and semi-structured data	yes	yes	no	yes
Query over heterogeneous data	yes	yes	no	yes
Query over XML data model	yes	yes	no	no
Navigation through hierarchical data structures (Path Expressions)	yes	yes	no	exact match only
Joins (combine multiple data sources into a single result)	yes	no	yes	no
Dynamic data integration from multiple heterog. sources such as databases, documents and remote services	yes	yes	no	no
Data restructuring patterns (e.g. SELECT-FROM-WHERE in SQL)	yes	no	yes	no
Iteration over sets (e.g. FOR clause)	yes	no	yes	no
General-purpose predicate expressions (WHERE clause)	yes	no	yes	no
Nesting several kinds of expressions with full generality	yes	no	no	no
Binding of variables and creating new structures from variables (LET clause)	yes	no	yes	no
Constructive queries	yes	no	no	no
Conditional expressions (IF ... THEN ... ELSE)	yes	no	yes	no
Arithmetic, comparison, logical and set expressions	yes, all	yes	yes, all	log. & string
Operations on data types from a type system	yes	no	yes	no
Quantified expressions (e.g. SOME, EVERY clause)	yes	no	yes	no
Standard functions for sorting, string, math, aggregation	yes	no	yes	no
User defined functions	yes	no	yes	no
Regular expression matching	yes	yes	no	no
Concise and easy to understand queries	yes	yes	yes	yes

Figure 4: Capabilities of XQuery, XPath, SQL and LDAP query languages.

[8] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.

[9] Nelson Minar. Peer-to-Peer is Not Always Decentralized. In *The O'Reilly Peer-to-Peer and Web Services Conference*, Washington, D.C., November 2001.

[10] Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int. IEEE Symposium on Parallel and Distributed Computing (ISPDC 2002) (to appear)*, Iasi, Romania, July 2002.

[11] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, September 2000.

[12] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, October 2000.

[13] Dan Suciu. On Database Theory and XML. *SIGMOD Record*, 30(3), 2001.

[14] Mary Fernandez, Morishima Atsuyuki, Dan Suciu, and Tan Wang-Chiew. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[15] Daniela Florescu, Ioana Manolescu, and Donald Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Int. Conf. on Very Large Data Bases (VLDB)*, Roma, Italy, September 2001.

[16] World Wide Web Consortium. XML Schema Part 0: Primer. *W3C Recommendation*, May 2001.

[17] World Wide Web Consortium. XML Query Use Cases. *W3C Working Draft*, December 2001.

[18] World Wide Web Consortium. XML Query Requirements. *W3C Working Draft*, February 2001.

[19] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators. *W3C Working Draft*, December 2001.

[20] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. *Lecture Notes in Computer Science*, 42(7), December 2000.

[21] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. *W3C Recommendation*, November 1999.

[22] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). In *The Query Languages Workshop (QL'98)*, Boston, Massachussets, December 1998.

[23] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Eighth Int. World Wide Web Conference*, 1999.

[24] International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.

[25] Rick Cattell et al. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann Publishers, San Francisco, 1996.

[26] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. A Grid Monitoring Architecture. Technical report, Grid Forum Working Draft GWD-Perf-16-2, January 2002. http://www.gridforum.org.

[27] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.

[28] International Telecommunications Union. Recommendation X.500, Information technology – Open System Interconnection – The directory: Overview of concepts, models, and services. *ITU-T*, November 1995.

[29] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int. Symposium on*

High-Performance Distributed Computing (HPDC-10), San Francisco, California, August 2001.

[30] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *6th Int. Symposium on High Performance Distributed Computing (HPDC '97)*, 1997.

[31] Steve Fisher et al. Information and Monitoring (WP3) Architecture Report. Technical report, DataGrid-03-D3.2, January 2001.

[32] W. P. Dinda and B. Plale. A Unified Relational Approach to Grid Information Services. Technical report, Grid Forum Informational Draft GWD-GIS-012-1, February 2001. http://www.gridforum.org.

[33] P. Mockapetris. Domain Names - Implementation and Specification. *IETF RFC 1035*, November 1987.

[34] Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.

[35] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[36] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical report, U.C. Berkeley UCB//CSD-01-1141, 2001.

[37] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[38] M. van Steen, P. Homburg, and A. Tanenbaum. A wide-area distributed system. *IEEE Concurrency*, 1999.

[39] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.

[40] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, July 1999.